A3: Rule-of-Three and the Trie

Re-submit Assignment

Due Sep 18 by 11:59pm **Points** 100 **Submitting** a file upload

Main Goal

In this assignment, you will practice with data structures and pointers in C++ by implementing a trie, a specialized tree structure that can have good performance re*trie*ving dictionary words. Your code will be evaluated not only for a functional trie, but also best practices with making classes that manage their own internal memory allocations.

Trie Background

A trie is an example of a generalized tree, in that the branching factor is not always two, like it is for binary trees, but potentially larger. In a classical trie, the branching factor is as big as the alphabet being used. For example, to represent hex digits, the branching factor would be 16, one possible branch for each of 0-F.

However, a branch represents a valid path through the tree for known stored values. So many possible branches would not exist in the tree - the branching is likely to be sparse. Each node represents a character of the alphabet, but that character is not stored in the node - if the program arrived at the node from an 'E' branch, then the node represents 'E'.

One way of representing the branching is a fixed size array with the letter implicit in the position in the array. This has fixed lookup cost for a letter. Another way would be storage that only holds valid branches and knows what letter the branch represents. For this assignment, you must represent the branching as an array of C-style pointers to nodes, where the array is the same size as the alphabet. Our alphabet will be the set of lower-case English letters, 'a' - 'z'.

You should be cautious with pointers. Pointers that are not pointing to a meaningful memory address should be set to nullptr. In a trie, a branching array element with a nullptr means that no word continues down that character branch.

Making a Trie Class

Your job is to make a Trie class. The Trie class should have a root node, where nodes have internal storage for branches of the Trie and a flag determining whether or not that node represents the end of a valid word. The class needs the following public interface:

1. A default constructor. A Trie should report that an empty key string is not in the Trie.
2. A destructor. The destructor should not have memory-leaks.
3. A copy constructor and an assignment= operator using the approaches from the Rule-Of-Three slides.
4. A method addWord that accepts a std::string and returns void. The word passed into the method should be added to the Trie. Duplicate adds should not change the Trie. You may assume that the word is only made up of lower-case characters from a-z.
5. A method isWord that accepts a std::string and returns bool. The argument is a string made up of characters 'a'-'z'. If the word is found in the Trie, the method should return true, otherwise return false.
6. A method allWordsStartingWithPrefix that accepts a std::string and returns a vector<std::string> that contains all the words in the Trie that start with the passed in argument prefix string. A word that is just the prefix is acceptable. An empty prefix should return all words in the Trie.

The Trie class should be in Trie.h and Trie.cpp files. You may decide to add other classes to assist your design of the Trie, but the interface to the Trie should have the above elements.

Recursion

Some parts of the Trie are elegantly solved with recursion. You will need to be cautious about passing some parameters by value and others by reference to either retain or lose changes during the recursion. I find I need to get in a recursive mind-set to write recursive code - try a simpler problem first.

Test Program

You should write a test program TrieTest.cpp that exercises your Trie. The test program should have the following specification:

1. It should take in two command-line arguments, each one a filename. The first is a file of words, each on their own line, with the words all lowercase and only made up of characters a-z. The second is a file of queries, each word on its own line and also of acceptable characters. Be careful with text file encodings. A text file written in Windows may use different end of line characters than Linux and may break the a-z alphabet when used on a different OS. You do not need to manage that yourself, but your testing may have weird problems if you are hopping machine types.
2. You should do error checking on the number of arguments and whether or not the file exists and quit gracefully if anything is wrong.
3. All the words in the first file should be added to a Trie.
4. All the words in the second file should be tested against the Trie using the isWord method. The result of the test should be reported to standard out (cout) in the following fashion.
    1. If the word is found, say "*word* is found", where *word* is the actual tested key.
    2. If the word is not found, say "*word* is not found, did you mean:" followed by a list of words in the dict that start with *word*, one alternate per line, with the listing indented 3 spaces. If no alternatives are found, instead report "no alternatives found", also indented 3 spaces and on its own line.

During development of the test program, you should write code that satisfies yourself that the Trie class is robust to memory leaks. You should look at tools like valgrind - **http://valgrind.org.** (http://valgrind.org.) which can help test memory usage. You may find that you need to artificially enforce your Trie variable to be destructed before the main function finishes - you can do that with an artificial {} block in main.

Make a directory A3 and put the source files and a Makefile in the directory. The Makefile should not use absolute paths, only relative ones (this means that if you relocate your directory, it should still make properly). The executable target should be TrieTest. Also have a 'clean' target that removes ./TrieTest and *.o files.

Use -std=c++11 and -Wall for compilation flags. Code will be tested on the lab2 machines. I have see problems where code on a Mac runs but crashes on Linux - maybe based on not initializing the node array with nullptrs properly.

Advanced

Make a Trie method wordsWithWildcardPrefix that accepts a std::string and returns a vector<std::string> that contains all the words in the Trie that start with the passed in argument prefix string. The prefix string can now contain 0 or more wildcard characters '?'. The wildcard character can stand in for any one valid letter or no letter. The returned list of words must be unique - so there are no repeated words in the list. This applies for the allWordsWithPrefix method above as well, but you shouldn't be running into that issue there.

This part of the assignment is worth 4% of the grade. If you do not wish to do it, just make a stub implementation that returns an empty vector. This method will likely depend some on iterators and generic programming, topics we will cover in the next lecture. You will likely be asking StackOverflow a number of questions on how to do certain manipulations of strings and vectors, and that is OK. Just remember to cite your source of techniques you learn from these sites. In case that confuses anyone, you may not look at full C++ Trie implementations and copy what is done for this assignment.

I spend 30 minutes or so fiddling with this one method, even with having to research 3 or 4 things I needed to do, so it is definitely doable.

Submission

Zip the directory and submit by the due date. Code should have a comment header with file and personal identifying information and comments should be used to explain less-readable code.

Peer Review

After the deadline, you will be asked to review two other sets of code.